Short Paper: Proposing a New Foundation of Attack Trees in Monoidal Categories

Harley Eades III

Computer Science
Augusta University
heades@augusta.edu

Abstract

This short paper introduces a new research direction studying at the intersection of threat analysis using attack trees and interactive theorem proving using linear logic. Currently, the project has developed a new semantics of attack trees in dialectica spaces, a well-known model of intuitionistic linear logic, which offers a new branching operator to attack trees. Then by exploiting the Curry-Howard-Lambek correspondence the project seeks to develop a new domain-specific linear functional programming language called Lina – for Linear Threat Analysis – for specifying and reasoning about attack trees.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures

General Terms TODO

Keywords TODO

1. Introduction

What do propositional logic, multisets, directed acyclic graphs, source sink graphs (or parallel-series pomsets), Petri nets, and Markov processes all have in common? They are all mathematical models of attack trees – see the references in (Kordy 2014; Jhawar 2015) – but also, they can all be modeled in some form of a symmetric monoidal category[1] (Tzouvaras 1998; Brown 1991; Fiore 2013; Francesco Albasini 2010) – for the definition of a symmetric monoidal category see Appendix A. Taking things a little bit further, monoidal categories have a tight correspondence with linear logic through the beautiful Curry-Howard-Lambek correspondence (Barr 1991). This correspondence states that objects of a monoidal category correspond to the formulas of linear logic and the morphisms correspond to proofs of valid sequents of the logic. I propose that attack trees – in many different flavors – be modeled as objects in monoidal categories, and hence, as formulas of linear logic.

Suppose we are the computer security team of a small hospital, and we need to asses the potential for an intruder to gain root access of the all important medical records Unix server. So we build an attack tree to asses all of the potential ways one could gain root access to the server. Such an attack tree might look something like the following:

```
⊔ "Obtain Root Privileges" 220
  (⊔ "Access System Console" 130
    (⊔ "Enter Computer Center" 110
      (leaf "Break In to Computer Center" 80)
      (leaf "Unattended Guest" 30))
    (leaf "Corrupt Operator" 20))
  (⊔ "Obtain Root Password" 90
    (▷ "Guess Password" 55
      (leaf "Obtain Password File" 15)
      (leaf "Encounter Guessable Password" 40))
    (⊔ "Look Over Sys. Admin. Shoulder" 35)
      (leaf "Trojan Horse Root" 25)
      (leaf "Corrupt Sys. Admin." 10))
```

Attack trees are a modeling tool, originally proposed by Bruce Schneier (**?**), which are used to assess the threat potential of a security critical system. Attack trees have since been used to analyze the threat potential of many types of security critical systems, for example, cybersecurity of power grids (**?**), wireless networks (**?**), and many others. Attack trees consists of several goals, usually specified in English prose, for example, "compromise safe" or "obtain administrative privileges", where the root is the ultimate goal of the attack and each node coming off of the root is a refinement of the main goal into a subgoal. Then each subgoal can be further refined. The leaves of an attack tree make up the set of base attacks. Subgoals can be either disjunctively or conjunctively combined.

There have been a number of extensions of attack trees to include new operators on goals. One such extension recasts attack trees into attack nets which have all of the benefits of attack trees with the additional benefit of being able to include the flaw hypothesis model for penetration testing (**?**). A second extension adds sequential conjunction of attacks, that is, suppose A_1 and A_2 are attacks, then $A_1; A_2$ is the attack obtained by performing A_1, and then executing attack A_2 directly after A_1 completes (Jhawar 2015).

Attack trees for real-world security scenarios can grow to be quite complex. The attack tree presented in (**?**) to access the security of power grids has twenty-nine nodes with sixty counter measures attached to the nodes throughout the tree. The details of the tree spans several pages of appendix. The attack tree developed for the border gateway protocol has over a hundred nodes (**?**), and the details of the tree spans ten pages. Manipulating such large trees without a formal semantics can be dangerous.

One of the leading questions the field is seeking to answer is "what is an attack tree?" That is, what is a mathematical founda-

[1] I provide a proof that the category of source sink graphs is monoidal in Appendix B.

PLAS '16 October 24, 2016, Vienna, Austria

ACM . . . $15.00

tion of attack trees? There have been numerous attempts at answering this question. For example, attack trees have been based on propositional logic and De Morgan Algebras (**???**), multisets (Mauw 2006), Petri nets (**?**), tree automata (**?**), and series parallel graphs (Jhawar 2015). **There is currently no known semantics of attack trees based in category theory**.

By far the most intuitive foundation of attack trees is propositional logic or De Morgan algebras, however, neither of these properly distinguish between attack trees with repeated subgoals. If we consider each subgoal as a **resource** then the attack tree using a particular resource twice is different than an attack tree where it is used only once. The multiset semantics of attack trees was developed precisely to provide a resource conscious foundation (Mauw 2006). The same can be said for the Petri nets semantics (**?**). A second benefit of a semantics based in multisets, Petri nets, and even tree automata is that operators on goals in attack trees are associated with concurrency operators from process algebra. That is, the goals of an attack tree should be thought of as being run concurrently. Furthermore, when moving to these alternate foundations **the intuitiveness and elegance of the propositional logic semantics is lost**. Lastly, existing work has focused on specifically what an attack tree is, and has not sought to understand what the theory of attack trees is.

By modeling attack trees in monoidal categories we obtain a sound mathematical model, a resource conscious logic for reasoning about attack trees, and the means of constructing a functional programming language for defining attack trees (as types), and constructing semantically valid transformations (as programs) of attack trees.

Linear logic was first proposed by Girard (Girad 1987) and was quickly realized to be a theory of resources. In linear logic, every hypothesis must be used exactly once. Thus, formulas like $A \otimes A$ and A are not logically equivalent – here $\otimes$ is linear conjunction. This resource perspective of linear logic has been very fruitful in computer science and lead to linear logic being a logical foundation of processes and concurrency where formulas may be considered as processes. Treating attack trees as concurrent processes is not new; they have been modeled by event-based models of concurrency like Petri nets and partially-ordered multisets (pomsets) (Jhawar 2015; Mauw 2006). In fact, pomsets is a model in which events (the resources) can be executed exactly once, and thus, has a relationship with linear logic (Retore 1997). However, connecting linear logic as a theory of attack trees is novel, and strengthens this perspective.

In this short paper I introduce a newly funded research project[2] investigating founding attack trees in monoidal categories, and through the Curry-Howard-Lambek correspondence deriving a new domain-specific functional programming language called Lina for Linear Threat Analysis. Note that this paper introduces an ongoing research project, and thus, we do not currently have the complete story, but we feel that the community will find this project of interest, and the project would benefit from the feedback of the community. I begin by defining an extension – inspired by our semantics – of the attack trees given in (Jhawar 2015) in Section 2. Then I introduce a new semantics of attack trees in dialectica spaces, which depends on a novel result on dialectica spaces, in Section 3. The final section, Section 4, discusses Lina and some of the current problems the project seeks to answer.

[2] This material is based upon work supported by the National Science Foundation CRII CISE Research Initiation grant, "CRII:SHF: A New Foundation for Attack Trees Based on Monoidal Categories", under Grant No. 1565557.

2. Attack Trees

In this paper I consider an extension of attack trees with sequential composition which are due to Jhawar et al. (Jhawar 2015), but one of the projects ultimate goals is to extend attack trees with even more operators driven by our choice of semantics. The syntax for attack trees is defined in the following definition.

Definition 1. *The following defines the syntax of **Attack Trees** given a set of base attacks $b \in \mathsf{B}$:*

$$t ::= b \mid t_1 \odot t_2 \mid t_1 \sqcup t_2 \mid t_1 \rhd t_2 \mid t_1 \otimes t_2$$

I denote unsynchronized non-communicating parallel composition of attacks by $t_1 \odot t_2$, choice between attacks by $t_1 \sqcup t_2$, sequential composition of attacks by $t_1 \rhd t_2$, and a new operator called unsynchronized interacting parallel composition denoted $t_1 \otimes t_2$.

The following rules define the attack tree equivalence relation:

$$\frac{}{(t_1 \,\mathsf{op}\, t_2) \,\mathsf{op}\, t_3 = t_1 \,\mathsf{op}(t_2 \,\mathsf{op}\, t_3)}\ \text{ASSOC}$$

$$\frac{}{t_1 \,\mathsf{op_S}\, t_2 = t_2 \,\mathsf{op_S}\, t_1}\ \text{SYM}$$

$$\frac{}{(t_1 \sqcup t_2) \odot t_3 = (t_1 \odot t_3) \sqcup (t_2 \odot t_3)}\ \text{DIST}_1$$

$$\frac{}{(t_1 \sqcup t_2) \rhd t_3 = (t_1 \rhd t_3) \sqcup (t_2 \rhd t_3)}\ \text{DIST}_2$$

where $\mathsf{op} \in \{\odot, \otimes, \rhd, \sqcup\}$ *and* $\mathsf{op_S} \in \{\odot, \otimes, \sqcup\}$.

The syntax given in the previous definition differs from the syntax used by Jhawar et al. (Jhawar 2015). First, I use infix binary operations, while they use prefix n-ary operations. However, it does not sacrifice any expressivity, because each operation is associative, and parallel composition, choice, and interacting parallel composition are symmetric. Thus, Jhawar et al.'s definition of attack trees can be embedded into the ones defined here.

The second major difference is that the typical parallel composition operator found in attack trees is modeled here by unsynchronized non-communicating parallel composition which happens to be a symmetric tensor product, and not a disjunction. This is contrary to the literature, for example, the parallel operation of Jhawar et al. defined on source sink graphs (Jhawar 2015) can be proven to be a coproduct – see Appendix B – and coproducts categorically model disjunctions. Furthermore, parallel composition is modeled by multiset union in the multiset semantics, but we can model this as a coproduct. However, in the semantics given in the next section if we took parallel composition to be a coproduct, then the required isomorphisms necessary to model attack trees would not exist.

The third difference is that I denote the choice between executing attack t_1 or attack t_2, but not both, by $t_1 \sqcup t_2$ instead of using a symbol that implies that it is a disjunction. This fits very nicely with the semantics of Jhawar et al., where they collect the attacks that can be executed into a set. The semantics I give in the next section models choice directly.

The fourth, and final, difference is that I extend the syntax with a new operator called unsynchronized communicating parallel composition. The attack $t_1 \otimes t_2$ states that t_1 interacts with the attack t_2 in the sense that processes interact. Modeling interacting attacks allows for the more refined modeling of security critical systems, for example, it can be used to bring social engineering into the analysis where someone communicates malicious information or commands to a unsuspecting party. As a second example, interacting parallel composition could be used to model interacting bot nets.

Finally, the equivalence relation is essentially the equivalence given in Jhaware et al. (Jhawar 2015) – Theorem 1.

3. Semantics of Attack Trees in Dialectica Spaces

I now introduce a new semantics of attack trees that connects their study with a new perspective of attack trees that could highly impact future research: intuitionistic linear logic, but it also strengthens their connection to process calculi. This section has been formalized in the proof assistant Agda[3]. The semantics is based on the notion of a dialectica space:

Definition 2. *A **dialectica space** is a triple (A, Q, δ) where A and Q are sets and $\delta : A \times Q \to \mathsf{3}$ is a multi-relation where $\mathsf{3} = \{0, \bot, 1\}$ and $\bot$ represents undefined.*

Dialectica spaces can be seen as the intuitionistic cousin (de Paiva 2006) of Chu spaces (Pratt 1999). The latter have be used extensively to study process algebra and as a model of classical linear logic, while dialectica spaces and their morphisms form a categorical model of intuitionistic linear logic called $\mathsf{Dial}_3(\mathsf{Sets})$ (originally due to (de Paiva 1989)); I do not introduce dialectica space morphisms here, but the curious reader can find the definition in the formal development. I will use the intuitions often used when explaining Chu spaces as processes to explain dialectica spaces as processes, but it should be known that these intuitions are due to Pratt and Gupta (Gupta 1994).

Intuitively, a dialectica space, (A, Q, δ), can be thought of as a process where A is the set of actions the process will execute, Q is the set of states the process can enter, and for $a \in A$ and $q \in Q$, $\delta(a, q)$ indicates whether action a can be executed in state q.

The interpretation of attack trees into dialectica spaces requires the construction of each operation on dialectica spaces:

Parallel Composition. Suppose $\mathcal{A} = (A, Q, \alpha)$ and $\mathcal{B} = (B, R, \beta)$ are two dialectica spaces. Then we can construct – due to de Paiva (de Paiva 2014) – the dialectica space $\mathcal{A} \odot \mathcal{B} = (A \times B, Q \times R, \alpha \odot \beta)$ where $(\alpha \odot \beta)((a, b), (q, r)) = \alpha(a, q) \otimes_3 \beta(b, r)$ and $\otimes_3$ is the symmetric tensor product definable on $\mathsf{3}$[4]. Thus, from a process perspective we can see that $\mathcal{A} \odot \mathcal{B}$ executes actions of $\mathcal{A}$ and actions of $\mathcal{B}$ in parallel. Parallel composition is associative and symmetric.

Choice. Suppose $\mathcal{A} = (A, Q, \alpha)$ and $\mathcal{B} = (B, R, \beta)$ are two dialectica spaces. Then we can construct the dialectica space $\mathcal{A} \sqcup \mathcal{B} = (A + B, Q + R, \alpha \sqcup \beta)$ where $(\alpha \sqcup \beta)(i, j) = \alpha(i, j)$ if $i \in A$ and $j \in Q$, $(\alpha + \beta)(i, j) = \beta(i, j)$ if $i \in B$ and $j \in R$, otherwise $(\alpha + \beta)(i, j) = 0$. Thus, from a process perspective we can see that $\mathcal{A} \sqcup \mathcal{B}$ executes either an action of $\mathcal{A}$ or an action of $\mathcal{B}$, but not both. Choice is symmetric and associative, but it is not a coproduct, because it is not possible to define the corresponding injections. Brown et. al. show that Petri nets can be modeled in dialectica spaces (Brown 1991), but they use the coproduct as choice. The operator given here is actually the definition given for Chu spaces (Gupta 1994). If we were to use the coproduct, then we would not be able to prove that choice distributes over parallel composition nor over sequential composition. As far as I am aware, this is the first time this has been pointed out.

Sequential Composition. Suppose $\mathcal{A} = (A, Q, \alpha)$ and $\mathcal{B} = (B, R, \beta)$ be two dialectica spaces. Then we can construct – due to de Paiva (de Paiva 2014) – the dialectica space $\mathcal{A} \rhd \mathcal{B} = (A \times B, Q \times R, \alpha \rhd \beta)$ where $(\alpha \rhd \beta)((a, b), (q, r)) = \alpha(a, q) \mathbin{\mathsf{land}} \beta(i, r)$, and land is lazy conjunction defined for $\mathsf{3}$[5]. This is a non-symmetric conjunctive operator, and thus, sequential composition is non-symmetric. This implies that from a process perspective $\mathcal{A} \rhd \mathcal{B}$ will first execute the actions of $\mathcal{A}$ and then execute actions of $\mathcal{B}$ in that order. Sequential composition is associative.

Interacting Parallel Composition. Suppose $\mathcal{A} = (A, Q, \alpha)$ and $\mathcal{B} = (B, R, \beta)$ are two dialectica spaces. Then we can construct the dialectica space $\mathcal{A} \otimes \mathcal{B} = (A \times B, (B \to Q) \times (A \to R), \alpha \otimes \beta)$ where $B \to Q$ and $A \to R$ denote function spaces, and $(\alpha \otimes \beta)((a, b), (f, g)) = \alpha(a, f(b)) \wedge \beta(b, g(a))$. From a process perspective the actions of $\mathcal{A} \otimes \mathcal{B}$ are actions from $\mathcal{A}$ and actions of $\mathcal{B}$, but the states are pairs of maps $f : B \to Q$ and $g : A \to R$ from actions to states. This is the point of interaction between the processes. This operator is symmetric and associative.

At this point it is straightforward to define an interpretation $[\![t]\!]$ of attack trees into $\mathsf{Dial}_3(\mathsf{Sets})$. Soundness with respect to this model would correspond to the following theorem.

Theorem 3 (Soundness)**.** *If $t_1 \rightsquigarrow t_2$, then $[\![t_1]\!]$ is isomorphic to $[\![t_2]\!]$ in $\mathsf{Dial}_3(\mathsf{Sets})$.*

Those familiar with Chu spaces and their application to process algebra may be wondering how treating dialectica spaces as processes differs. The starkest difference is that in this model process simulation is modeled by morphisms of the model, but this is not possible in Chu spaces. In fact, to obtain the expected properties of processes a separate notion of bi-simulation had to be developed for Chu spaces (Gupta 1994). However, I took great care to insure that the morphisms of our semantics capture the desired properties of process simulation, and hence, attack trees.

The ability to treat morphisms as process simulation was not easy to achieve. The definition of choice in the semantics presented here actually is the definition given for Chu spaces (Gupta 1994), but Brown et al. use the coproduct defined for dialectica spaces to model choice in Petri nets. However, taking the coproduct for choice here does not lead to the isomorphisms $(\mathcal{A} \sqcup \mathcal{B}) \rhd \mathcal{C} \cong (\mathcal{A} \rhd \mathcal{C}) \sqcup (\mathcal{B} \rhd \mathcal{C})$ and $(\mathcal{A} \sqcup \mathcal{B}) \odot \mathcal{C} \cong (\mathcal{A} \odot \mathcal{C}) \sqcup (\mathcal{B} \odot \mathcal{C})$, thus, we will not be able to soundly model attack trees. I have found that if choice is modeled using the definition from Chu spaces (Gupta 1994) then we obtain these isomorphisms which is a novel result[6].

This semantics can be seen as a generalization of some existing models. Multisets, pomsets, and Petri nets can all be modeled by dialectica spaces (Brown 1991; Gupta 1994). However, there is a direct connection between dialectica spaces and linear logic which may lead to a logical theory of attack trees.

4. Lina: A Domain Specific PL for Threat Analysis

The second major part of this project is the development of a staticly-typed domain-specific linear functional programming language for specifying and reasoning about attack trees called Lina for Linear Threat Analysis. Lina will consist of a core language and a surface language. We view attack trees as consisting of two layers: a logic layer and a quantitative layer. The former is described by the definition of attack trees in the previous section, but the latter is the layer added atop of the logical layer used when conducting

[3] The complete formalization can be found at `https://github.com/heades/dialectica-spaces/tree/PLAS16` which is part of a general library for working with dialectica spaces in Agda developed with Valeria de Paiva.

[4] See the formal development for the full definition: `https://github.com/heades/dialectica-spaces/blob/PLAS16/concrete-lineales.agda#L328`

[5] See the formal development for the full definition: `https://github.com/heades/dialectica-spaces/blob/PLAS16/concrete-lineales.agda#L648`

[6] For the proofs see the formal development: `https://github.com/heades/dialectica-spaces/blob/PLAS16/concurrency.agda#L70` and `https://github.com/heades/dialectica-spaces/blob/PLAS16/concurrency.agda#L150`

analysis, for example, computing set of attacks with minimal cost. Thus, there are two types of proofs about attack trees. Proofs about the logical layer will be checked using a linear type system, but proofs about the quantitative layer will be mostly numerical. In this section we largely concentrate on the logical layer which is the current focus of the project.

Lina's core will consist of a language for defining attack trees, and this language will consist of the two layers, but one benefit of the layered view of attack trees is that the logical layer can be projected out, and hence, type checking may completely ignore the quantitative layer. The following two sections describe both of these concepts.

4.1 Lina's Core: Defining Attack Trees

Consider the attack tree for assessing the risk of becoming root on a Unix machine from above. We can represent that tree as follows in the form of a functional program:

$$
\begin{array}{l}
\sqcup\text{"Obtain Root Privileges"}\ r \\
\quad (\sqcup\text{"Access System Console"}\ r \\
\qquad (\sqcup\text{"Enter Computer Center"}\ r \\
\qquad\quad (\mathsf{leaf}\ \text{"Break In to Computer Center"}\ 80) \\
\qquad\quad (\mathsf{leaf}\ \text{"Unattended Guest"}\ 30)) \\
\qquad (\mathsf{leaf}\ \text{"Corrupt Operator"}\ 20)) \\
\quad (\sqcup\text{"Obtain Root Password"}\ r \\
\qquad (\triangleright\text{"Guess Password"}\ r \\
\qquad\quad (\mathsf{leaf}\ \text{"Obtain Password File"}\ 25) \\
\qquad\quad (\mathsf{leaf}\ \text{"Encounter Guessable Password"}\ 10)) \\
\qquad (\sqcup\text{"Look Over Sys. Admin. Shoulder"}\ r) \\
\qquad\quad (\mathsf{leaf}\ \text{"Trojan Horse Root"}\ 15) \\
\qquad\quad (\mathsf{leaf}\ \text{"Corrupt Sys. Admin."}\ 40))
\end{array}
$$

where $r = (\lambda x.\lambda y.x\ +\ y)$.

We can see from this example that we treat the nodes of the attack tree as combinators, either $\mathsf{leaf}\ l\ q$ or $c\ l\ q$, where c is a branching node symbol, l is a label, for example, a string, and q is a quantitative expression. The types of l and q depend on the type of the tree itself, for example, the type of the tree above is $\mathsf{AttackTree\ String\ Double}$, and thus, labels are strings, and leafs are constant doubles, but the quantitative data on branching nodes has type $\mathsf{Double} \to \mathsf{Double} \to \mathsf{Double}$. Thus, one novelty of Lina is that data at nodes can be higher order, but the data at branching nodes is always a binary function whose first argument is the data from the left tree, and the second argument is the data from the right tree. Hence, making it easier and more precise to compute costs across the tree. In the example above we simply just sum the data, but one could expect more complex polynomials being useful for analysis. Finally, branching nodes have two additional arguments, t_1 and t_2, which are the left and right trees.

Throughout the remainder of this section we give a brief overview of the preliminary design of Lina's language for defining attack trees. The following defines the syntax (d ranges over any double, and s ranges over any string):

$$
\begin{array}{rl}
\text{(Quantitative Types)} & Q := \mathsf{Double} \mid Q \to Q \\
\text{(Numeric Operators)} & \mathsf{op} := + \mid - \mid * \mid / \\
\text{(Quantitative Expressions)} & q := x \mid d \mid \lambda x.q \mid q_1\ q_2 \mid \\
 & \qquad q_1\ \mathsf{op}\ q_2 \mid \mathsf{rec}\ q_0\ \mathsf{of}\ q_1, q_2 \\
\text{(Label Types)} & L := \mathsf{String} \mid \mathsf{Double} \\
\text{(Labels)} & l := s \mid d \\
\text{(Kinds)} & k := \mathsf{AttackTree}\ L\ Q \mid k_1 \to k_2 \\
\text{(Attack Tree Combinators)} & c := \otimes \mid \odot \mid \triangleright \mid \sqcup \\
\text{(Attack Trees)} & t := x \mid \lambda x.t \mid t_1\ t_2 \mid \mathsf{leaf}\ l\ q \mid \\
 & \qquad c\ l\ q
\end{array}
$$

Typing for this language is straightforward, and to save space we do not give every rule. The typing rules for the quantitative language corresponds to the simply typed λ-calculus with doubles, numeric operators, and a recursor in the spirit of Gödel's system T, and thus, we omit their rules here, but denote the typing judgment by $\Delta \vdash q : Q$, where Δ is a typing context consisting of pairs $x : Q$; for the complete set of rules see Appendix **??**. Typing labels is trivial, and we denote the typing judgment by $\vdash\ l\ :\ L$. Finally, the following rules defines the kinding rules for attack trees:

$$\frac{}{\Gamma_0, x : k, \Gamma_1 \vdash x : k}\ \text{K\_VAR} \qquad \frac{\Gamma, x : k_0 \vdash t : k_1}{\Gamma \vdash \lambda x.t : k_0 \to k_1}\ \text{K\_FUN}$$

$$\frac{\Gamma \vdash t_1 : k_0 \quad \Gamma \vdash t_0 : k_0 \to k_1}{\Gamma \vdash t_0\ t_1 : k_1}\ \text{K\_APP}$$

$$\frac{\vdash l : L \quad \cdot \vdash q : Q}{\Gamma \vdash \mathsf{leaf}\ l\ q : \mathsf{AttackTree}\ L\ Q}\ \text{K\_LEAF}$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 : \mathsf{AttackTree}\ L\ Q \quad \vdash l : L \\ \Gamma \vdash t_2 : \mathsf{AttackTree}\ L\ Q \quad \cdot \vdash q : Q \to Q \to Q\end{array}}{\Gamma \vdash c\ l\ q\ t_1\ t_2 : \mathsf{AttackTree}\ L\ Q}\ \text{K\_COMB}$$

Functions over attack trees will allow for the definition of attack tree schemas that one could use to build up a library of attack trees. At this point one could speak of evaluating attack trees which would correspond to normalizing the tree to its combinator form, but we could also speak about evaluating the data of the tree, but how this is done is left for future work.

Lina's attack tree definition language technically lives at the type level. We can see that the logical layer of each attack tree corresponds to a linear type. When attack trees get large it makes sense to want to restructure the tree to gain new insights, but existing tools do not support this in such a way that one knows that the tree obtained after restructuring is semantically equivalent to the original tree. Due to the Curry-Howard-Lambek correspondence Lina will come equipped with a linear type system that can be used to register semantically valid transformations of attack trees as programs between linear types, but there is a big hurdle that first must be crossed.

4.2 Lina's Core: A Linear Type System

The current main focus of the project is the design and analysis of Lina's core type system. However, there is a significant hurdle we must get through first. Types in Lina will correspond to attack trees while programs correspond to semantically valid transformations of attack trees, thus, a question we must answer then is **how do we sufficiently represent the model of attack trees in $\mathsf{Dial}_3(\mathsf{Sets})$ as a linear logic?** The problem is the fact that Lina will require both commutative (parallel composition and choice) and non-communicative monoidal operators (sequencing).

Supporting both commutative and non-communicative operators within the same linear logic has been a long standing question. A starting point might be with Reedy's LLMS which has already been shown to have a categorical model in $\mathsf{Dial}_3(\mathsf{Sets})$ by de Paiva (de Paiva 2014). In fact, the definition of non-interacting parallel composition given here is due to her model. However, we have taken a new path which we also approach categorically, and then syntactically.

We have obtained a great insight from our model in dialectica categories, but it is not obvious how to abstract it into a model we can easily extract a type system from. So instead of working directly with the dialectica model we have found a more abstract model with the ability to have both a commutative tensor product and and non-communicative tensor product within the same model, but is more straightforwardly transformed into a type system.

Girard's genius behind linear logic was that he isolated the structural rules – weakening and contraction – by treating them as

an effect and putting them inside a comonad called the of-course exponential denoted $!A$. In fact, $!A \otimes !A$ is logically equivalent to $!A$, and thus, by staying in the comonad we become non-linear. However, in Girard's linear logic exchange was not isolated like weakening and contraction. Thus, the tensor product was symmetric.

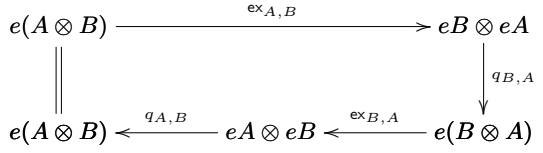
To accommodate both a commutative and non-commutative tensor product we isolate exchange in the same way that Girard isolated weakening and contraction. In this section I give a brief overview of how this is done. We begin with the notion of a Lambek category.

Definition 4. *A **Lambek category** is a monoidal category $(\mathcal{C}, I, \triangleright, \alpha, \lambda, \rho)$ where $\triangleright : \mathcal{C} \times \mathcal{C} \longrightarrow \mathcal{C}$ is the non-commutative tensor product, and $\alpha_{A,B,C} : (A \triangleright B) \triangleright C \longrightarrow A \triangleright (B \triangleright C)$, $\lambda_A : I \triangleright A \longrightarrow A$, and $\rho_A : A \triangleright I \longrightarrow A$ are all natural transformations subject to several coherence diagrams*[7].

We call the previous category a Lambek category to pay homage to Joachim Lambek and his work on the Lambeck calculus (**?**) which is a non-commutative substructural logic that predates linear logic. The traditional definition of a Lambek category also requires that the monoidal category be biclosed, but we do not concern ourselves here with closed categories.

In our model exchange will be considered as an effect, and so we isolate it inside a comonad. This will allow for the definition of a commutative tensor product.

Definition 5. *A **Lambek category with exchange** is a Lambek category $(\mathcal{C}, I, \triangleright, \alpha, \lambda, \rho)$ equipped with a monoidal comonad (e, ε, δ) where $e : \mathcal{C} \longrightarrow \mathcal{C}$ is a monoidal endofunctor*[8]*, and $\varepsilon_A : eA \longrightarrow A$ and $\delta_A : eA \longrightarrow e^2A$ are natural transformations. In addition, there is a natural transformation $\mathsf{ex}_{A,B} : e(A \triangleright B) \longrightarrow eB \triangleright eA$. Each of these morphisms are subject to the several coherence diagrams which we omit due to space. Most importantly, the following diagram must commute:*



The previous diagram can be seen as a form of invertibility for $\mathsf{ex}_{A,B}$.

The previous definition is largely based on how weakening and contraction are modeled by the of-course exponential, and how exchange operates in the coKleisli category of (e, ε, δ). The coKleisli category contains as objects all of the objects of $\mathcal{C}$, but has as morphisms all the morphisms of $\mathcal{C}$ whose source is of the form eA for some A. The coKleisli category is best viewed as the world inside a comonad. That is, it contains all of the structure of the ambient category, but also the additional effects the comonad provides. Thus, the coKleisli category of the exchange comonad should be a symmetric monoidal category, and indeed it is.

Lemma 6. *Suppose $\mathcal{C}$ is a Lambek category with exchange. Then the coKleisli category of the exchange comonad is symmetric monoidal.*

It is now should be more straightforward to construct a term assignment from this model. The following rules define a preliminary definition of a natural deduction term assignment:

$$\frac{}{x : T \vdash x : T}\ \text{VAR} \qquad \frac{}{\cdot \vdash \mathsf{unit} : \mathsf{I}}\ \text{UNITI}$$

$$\frac{\Gamma_2 \vdash s_1 : \mathsf{I} \quad \Gamma_1 \vdash s_2 : T}{\Gamma_1, \Gamma_2 \vdash \mathsf{let\ unit} = s_1 \ \mathsf{in}\ s_2 : T}\ \text{UNITE}$$

$$\frac{\Gamma_1 \vdash s_1 : T_1 \quad \Gamma_2 \vdash s_2 : T_2}{\Gamma_1, \Gamma_2 \vdash s_1 \otimes s_2 : T_1 \otimes T_2}\ \text{TENI}$$

$$\frac{\Gamma_2 \vdash s_1 : T_1 \otimes T_2 \quad \Gamma_1, x : T_1, y : T_2, \Gamma_3 \vdash s_2 : T_3}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathsf{let}\ x \otimes y = s_1 \ \mathsf{in}\ s_2 : T_3}\ \text{TENE}$$

$$\frac{\Gamma_2 \vdash s_1 : \mathsf{e}\ T_1 \otimes \mathsf{e}\ T_2 \quad \Gamma_1, w : \mathsf{e}\ T_2, x : \mathsf{e}\ T_1, \Gamma_2 \vdash s : T}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathsf{ex}\ s_1\ \mathsf{with}\ w, x\ \mathsf{in}\ s : T}\ \text{EX}$$

$$\frac{\Gamma \vdash s : \mathsf{e}\ T}{\Gamma \vdash \mathbf{derelict}\ s : T}\ \text{DERELICTION}$$

$$\frac{\overrightarrow{\Gamma_i \vdash s_i : \mathsf{e}\ T_i} \quad x_1 : \mathsf{e}\ T_1, \ldots, x_i : \mathsf{e}\ T_i \vdash s : T}{\Gamma \vdash \mathsf{promote}\ \overrightarrow{s}\ \mathsf{for}\ \overrightarrow{x_i}\ \mathsf{in}\ s : T}\ \text{PROMOTION}$$

The previous rules are based on the natural deduction formalization of intuitionistic linear logic due to (**?**).

We can interpret both sequential and interacting parallel composition as types. That is, we can interpret $t_1 \triangleright t_2$ as $T_1 \otimes T_2$, and $t_1 \otimes t_2$ as $\mathsf{e}\ T_1 \otimes \mathsf{e}\ T_2$. However, accommodating the other attack tree branching connectives in linear logic is left for future work.

4.3 Lina's Surface Language

Another goal of this project is to make using Lina as close as possible to functional programming as usual to prevent a large overhead of using the language as well as the tool. As a programming language simplicity is of the utmost importance, and I think with the semantics given here Lina will not require very advanced syntactic features. This cannot be said for some of the existing work that is similar to Lina. For example, Vigo et al. (Vigo 2014) proposed the Quality Tree Generator which requires the user to program in a process calculus which is a non-trivial overhead. At the tool level the goal is to have a completely graphical environment for creating attack trees and reasoning about them by capitalizing on existing graphical reasoning tools from category theory. Thus, at the tool level the user will not have to write any programs at all unless they want to extend the environment.

4.4 Automation

I do not consider this project, particularly Lina, to be at odds with existing work on using automated theorem proving to synthesize and analyze attack trees; see for example (Huistra 2016; Sheyner 2002; Vigo 2014; Wolters 2016). In fact, this project can benefit from automated generation of attack trees. Lina's primary goal is to make reasoning about attack trees safer by having a tight correspondence with the semantics of attack trees, and thus, will allow and help with the creation of attack trees. Lina will offer a manual way for one to create an attack tree, but by leveraging this existing work could allow for their automatic generation, but then could be used to restructure the tree and conduct further analysis in a semantically valid fashion. In addition, Lina can be seen as an interactive theorem prover for attack trees, and so could be used as a proof checker (Stump 2012) for proof producing SMT backed automated generation of attack trees, thus, potentially allowing for some of the analysis of attack trees in Lina to be automated.

[7] The coherence diagrams are equivalent to the ones in the definition of a symmetric monoidal category modulo symmetry; see the appendix for the complete definition of a symmetry monoidal category.

[8] For the full definition of a monoidal functor see Definition **??** in Appendix **??**.

5. Conclusion and Future Work

The project described here is to first develop the semantics of attack trees (Section 2) in dialectica spaces (Section 3), a model of full intuitionistic linear logic, and then exploiting the Curry-Howard-Lambek correspondence to develop a new functional programming language called Lina (Section 4) to be used to develop a new tool to conduct threat analysis using attack trees. This tool will include the ability to design and formally reason about attack trees using interactive theorem proving.

The core language will include a decidable type checker using term annotations on types. Programming with annotations can be very cumbersome, and so the surface language will use local type inference (Pierce 2000) to alleviate some of the burden from annotations.

References

Michael Barr. *-autonomous categories and linear logic. *Mathematical Structures in Computer Science*, 1:159–178, 7 1991.

Carolyn Brown, Doug Gurr, and Valeria de Paiva. A linear specification language for petri nets. *DAIMI Report Series*, 20(363), 1991.

Valeria de Paiva. Dialectica categories. In J. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, volume 92, pages 47–62. Amerian Mathemaitcal Society, 1989.

Valeria de Paiva. Dialectica and chu constructions: Cousins? *Theory and Applications of Categories*, 17(7):127–152, 2006.

Valeria de Paiva. Linear logic model of state revisited. *Logic Journal of IGPL*, 22(5):791–804, 2014.

Marcelo Fiore and Marco Devesas Campos. *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky: Essays Dedicated to Samson Abramsky on the Occasion of His 60th Birthday*, chapter The Algebra of Directed Acyclic Graphs, pages 37–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

Luisa Francesco Albasini, Nicoletta Sabadini, and Robert F. C. Walters. The compositional construction of markov processes. *Applied Categorical Structures*, 19(1):425–437, 2010.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987.

Vineet Gupta. *Chu Spaces: a Model of Concurrency*. PhD thesis, Stanford University, 1994.

D.J. Huistra. Automated generation of attack trees by unfolding graph transformation systems, March 2016.

Ravi Jhawar, Barbara Kordy, Sjouke Mauw, SaÅ!'a RadomiroviÄ, and Rolando Trujillo-Rasua. Attack trees with sequential conjunction. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection*, volume 455 of *IFIP Advances in Information and Communication Technology*, pages 339–353. Springer International Publishing, 2015.

Barbara Kordy, Ludovic Piétre-Cambacédés, and Patrick Schweitzer. Dag-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer Science Review*, 13â14:1 – 38, 2014.

Sjouke Mauw and Martijn Oostdijk. Foundations of attack trees. In DongHo Won and Seungjoo Kim, editors, *Information Security and Cryptology - ICISC 2005*, volume 3935 of *Lecture Notes in Computer Science*, pages 186–198. Springer Berlin Heidelberg, 2006.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000.

Vaughan Pratt. Chu spaces. Notes for the School on Category Theory and Applications University of Cimbra, July 1999.

Christian Retoré. *Typed Lambda Calculi and Applications: Third International Conference on Typed Lambda Calculi and Applications TLCA '97 Nancy, France, April 2–4, 1997 Proceedings*, chapter Pomset logic: A non-commutative extension of classical linear logic, pages 300–318. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

Peter Selinger. A survey of graphical languages for monoidal categories. *ArXiv e-prints*, August 2009.

Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 273–, Washington, DC, USA, 2002. IEEE Computer Society.

Aaron Stump, Andrew Reynolds, Cesare Tinelli, Austin Laugesen, Harley D. Eades III, Corey Oliver, and Ruoyu Zhang. Lfsc for smt proofs: Work in progress. In *Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving (PXTP 2012)*, 2012.

A Tzouvaras. The linear logic of multisets. *Logic Journal of IGPL*, 6(6):901–916, 1998.

R. Vigo, F. Nielson, and H. R. Nielson. Automated generation of attack trees. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, pages 337–350, July 2014.

N.H. Wolters. Analysis of attack trees with timed automata (transforming formalisms through metamodeling), March 2016.

Appendix

A. Symmetric Monoidal Categories

This appendix provides the definitions of both categories in general, and, in particular, symmetric monoidal closed categories. We begin with the definition of a category:

Definition 7. *A **category**, $\mathcal{C}$, consists of the following data:*

- *A set of objects $\mathcal{C}_0$, each denoted by A, B, C, etc.*
- *A set of morphisms $\mathcal{C}_1$, each denoted by f, g, h, etc.*
- *Two functions* src*, the source of a morphism, and* tar*, the target of a morphism, from morphisms to objects. If $\mathsf{src}(f) = A$ and $\mathsf{tar}(f) = B$, then we write $f : A \to B$.*
- *Given two morphisms $f : A \to B$ and $g : B \to C$, then the morphism $f; g : A \to C$, called the composition of f and g, must exist.*
- *For every object $A \in \mathcal{C}_0$, the there must exist a morphism $\mathsf{id}_A : A \to A$ called the identity morphism on A.*
- *The following axioms must hold:*
  - *(Identities) For any $f : A \to B$, $f; \mathsf{id}_B = f = \mathsf{id}_A; f$.*
  - *(Associativity) For any $f : A \to B$, $g : B \to C$, and $h : C \to D$, $(f; g); h = f; (g; h)$.*

Categories are by definition very abstract, and it is due to this that makes them so applicable. The usual example of a category is the category whose objects are all sets, and whose morphisms are set-theoretic functions. Clearly, composition and identities exist, and satisfy the axioms of a category. A second example is pre-ordered sets, $(A, \leq)$, where the objects are elements of A and a morphism $f : a \to b$ for elements $a, b \in A$ exists iff $a \leq b$. Reflexivity yields identities, and transitivity yields composition.

Symmetric monoidal categories pair categories with a commutative monoid like structure called the tensor product.

Definition 8. *A **symmetric monoidal category (SMC)** is a category, $\mathcal{M}$, with the following data:*

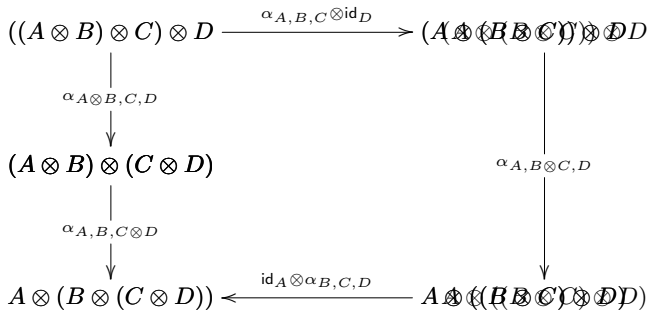
- *An object I of $\mathcal{M}$,*
- *A bi-functor $\otimes : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$,*
- *The following natural isomorphisms:*

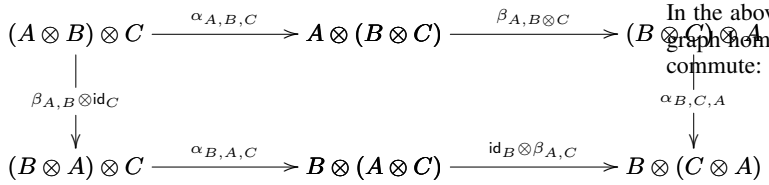
$$\lambda_A : I \otimes A \to A$$
$$\rho_A : A \otimes I \to A$$
$$\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$$

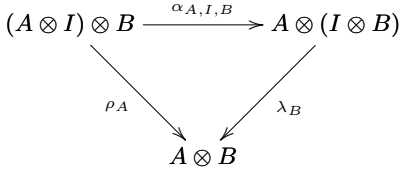
- *A symmetry natural transformation:*

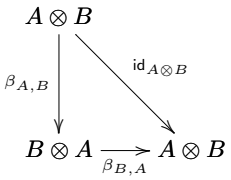
$$\beta_{A,B} : A \otimes B \to B \otimes A$$

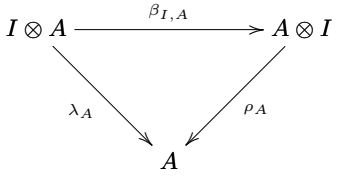
- *Subject to the following coherence diagrams:*











B. Source Sink Graphs are Symmetric Monoidal

In this appendix I show that the category of source-sink graphs defined by Jhawar et al. (Jhawar 2015) is symmetric monoidal. First, recall the definition of source-sink graphs and their homomorphisms.

Definition 9. *A **source-sink graph** over* B *is a tuple* $G = (V, E, s, z)$*, where* V *is the set of vertices,* E *is a multiset of labeled edges with support* $E^* \subseteq V \times \mathsf{B} \times V$*,* $s \in V$ *is the unique start,* $z \in V$ *is the unique sink, and* $s \neq z$*.*

Suppose $G = (V, E, s, z)$ *and* $G' = (V', E', s', z')$*. Then a **morphism between source-sink graphs**,* $f : G \to G'$*, is a graph homomorphism such that* $f(s) = s'$ *and* $f(z) = z'$*.*

Suppose $G = (V, E, s, z)$ and $G' = (V', E', s', z')$ are two source-sink graphs. Then given the above definition it is possible to define sequential and non-communicating parallel composition of source-sink graphs where I denote disjoint union of sets by $+$ (p 7. (Jhawar 2015)):

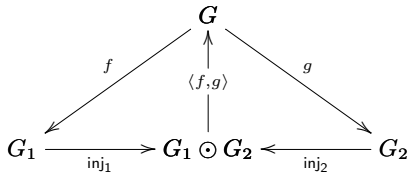
$$\text{(Sequential Composition)} \quad G \rhd G' = ((V \setminus \{z\}) + V', E^{[s'/z]} + E', s, z')$$
$$\text{(Parallel Composition)} \quad G \odot G' = ((V \setminus \{s, z\}) + V', E^{[s'/s, z'/z]} + E', s'$$

It is easy to see that we can define a category of source-sink graphs and their homomorphisms. Furthermore, it is a symmetric monoidal category were parallel composition is the symmetric tensor product. It is well-known that any category with co-products is symmetric monoidal where the co-product is the tensor product.

I show here that parallel composition defines a co-product. This requires the definition of the following morphisms:

$$\mathsf{inj}_1 : G_1 \to G_1 \odot G_2$$
$$\mathsf{inj}_2 : G_2 \to G_1 \odot G_2$$
$$\langle f, g \rangle : G_1 \odot G_2 \to G$$

In the above $f : G_1 \to G$ and $g : G_2 \to G$ are two source-sink graph homomorphisms. Furthermore, the following diagram must commute:



Suppose $G_1 = (V_1, E_1, s_1, z_1)$, $G_2 = (V_2, E_2, s_2, z_2)$, and $G = (V, E, s, z)$ are source-sink graphs, and $f : G_1 \to G$ and $g : G_2 \to G$ are source-sink graph morphisms – note that $f(s_1) = g(s_2) = s$ and $f(z_1) = g(z_2) = z$ by definition. Then we define the required co-product morphisms as follows:

$$\mathsf{inj}_1 : V_1 \to (V_1 \setminus \{s_1, z_1\}) + V_2 \qquad \mathsf{inj}_2 : V$$
$$\mathsf{inj}_1(s_1) = s_2 \qquad \mathsf{inj}_2(v)$$
$$\mathsf{inj}_1(z_1) = z_2$$
$$\mathsf{inj}_1(v) = v, \text{otherwise}$$

$$\langle f, g \rangle : (V_1 \setminus \{s_1, z_1\}) + V_2 \to V$$
$$\langle f, g \rangle(v) = f(v), \text{ where } v \in V_1$$
$$\langle f, g \rangle(v) = g(v), \text{ where } v \in V_2$$

It is easy to see that these define graph homomorphisms. All that is left to show is that the diagram from above commutes:

$$\begin{aligned} (\mathsf{inj}_1; \langle f, g \rangle)(s_1) &= \langle f, g \rangle(\mathsf{inj}_1(s_1)) \\ &= g(s_2) \\ &= s \\ &= f(s_1) \end{aligned} \qquad \begin{aligned} (\mathsf{inj}_1; \langle f, g \rangle)(z_1) &= \langle f, g \\ &= g(z_2 \\ &= z \\ &= f(z_1 \end{aligned}$$

Now for any $v \in V_1$ we have the following:

$$\begin{aligned} (\mathsf{inj}_1; \langle f, g \rangle)(v) &= \langle f, g \rangle(\mathsf{inj}_1(v)) \\ &= f(v) \end{aligned}$$

The equation for inj_2 is trivial, because inj_2 is the identity.